A Contract Based System For Large Data Visualization *

Hank Childs†
University of California, Davis/Lawrence Livermore National Laboratory
Eric Brugger, Kathleen Bonnell, Jeremy Meredith, Mark Miller, and Brad Whitlock ‡
Lawrence Livermore National Laboratory
Nelson Max§
University of California, Davis

ABSTRACT

VisIt is a richly featured visualization tool that is used to visualize some of the largest simulations ever run. The scale of these simulations requires that optimizations are incorporated into every operation VisIt performs. But the set of applicable optimizations that VisIt can perform is dependent on the types of operations being done. Complicating the issue, VisIt has a plugin capability that allows new, unforeseen components to be added, making it even harder to determine which optimizations can be applied.

We introduce the concept of a *contract* to the standard data flow network design. This contract enables each component of the data flow network to modify the set of optimizations used. In addition, the contract allows for new components to be accommodated gracefully within VisIt's data flow network system.

Keywords: large data set visualization, data flow networks, contract-based system

1 INTRODUCTION

VisIt is an end-user visualization and data analysis tool for diverse data sets, designed to handle data sets from thousands to millions to billions of elements in a single time step. The tool has a rich feature set; there are many options to subset, transform, render, and query data. VisIt has a distributed design. A server utilizes parallel compute resources for data reduction, while a client runs on a local desktop machine to maintain interactivity. The rendering primitives resulting from the data reduction phase are typically transferred to the client and rendered using graphics hardware. When the number of primitives overwhelms the client, the geometry is kept on the server and rendered using a sort-last rendering technique [9]. VisIt's rendering phase is outside the scope of this paper. Instead, we will focus on the data reduction phase and the optimizations necessary to handle large data sets.

VisIt employs a modified data flow network design [13] [1] [11]. Its base types are *data objects* and *components* (sometimes called process objects). The components can be *filters*, *sources*, or *sinks*. Filters have an input and an output, both of which are data objects. Sources have only data object outputs and sinks have only data object inputs. A *pipeline* is a collection of components. It has a source (typically a file reader) followed by many filters followed by a sink (typically a rendering engine). Pipeline execution is demand driven, meaning that data flow starts with a *pull* operation. This begins at the sink, which generates an *update* request that propagates up the pipeline through the filters, ultimately going to a load balancer (needed to divide the work on the server) and then to the source. The source generates the requested data which becomes input to the first filter. Then *execute* phases propagate down the pipeline. Each component takes the data arriving at its input, performs some operation and creates new data at its output until the sink is reached. These operations are typical of data flow networks. VisIt's data flow network design, is unique, however, in that it also includes a *contract* which travels up the pipeline along with update requests (see Figure 1).

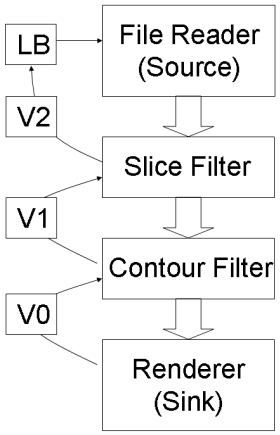

Figure 1: An example pipeline. During the update phase (denoted by thin arrows), Contract Version 0 (V0), comes from the sink. V0 is then an input to the contour filter, which modifies the contract to make Contract Version 1 (V1). This continues up the pipeline, until an executive that contains a load balancer (denoted by LB) is reached. This executive decides the details of the execution phase and passes those details to the source, which begins the execute phase (denoted by thick arrows).

VisIt is a large and complex system. It contains over 400 different types of components and data objects. It has over one million lines of source code and depends on many third party libraries. In addition, VisIt has a plugin capability that allows users to extend the tool with their own sources, sinks, filters, and even data objects.

The scale of the data sets processed by VisIt mandates that optimizations are incorporated into every pipeline execution. These optimizations vary from minimizing the data read in off disk to the treatment of that data to the way that data moves through a pipeline. The set of applicable optimizations is dependent on the properties of the pipeline components. This requires a dynamic system that determines which optimizations can be applied. Further, because of VisIt's plugin architecture, this system must be able to handle the addition of new, unforeseen components. VisIt's strategy is to have all of a pipeline's components modify a contract and have optimizations adaptively employed based on the specifications of this contract.

The heart of VisIt's contract-based system is an interface that allows pipeline components to communicate with other filters and describe their *impact* on a pipeline. Focusing on the more abstract notion of *impact* rather than the specifics of individual components allows VisIt to be a highly extensible architecture, because new components simply must be able to describe what *impacts* they will have. This abstraction also allows for effective management of the large number of existing components.

Because the contract is coupled with update requests, the information in the contract travels from the bottom of the pipeline to the top. When visiting each component in the pipeline, the contract

*This is LLNL Report UCRL-CONF-211357.
†e-mail: childs3@llnl.gov
‡e-mail: {brugger1,bonnell2,meredith6,miller86,whitlock2}@llnl.gov
§e-mail: max@cs.ucdavis.edu

is able to inform that component of the downstream components' requirements, as well as being able to guarantee that the components upstream will receive the current component's requirements. Further, the contract-based system enables all components to participate in the process of adaptively selecting and employing appropriate optimizations. Finally, combining the contract with *update* requests has allowed for seamless integration into VisIt.

2 Background

2.1 Description of Input Data

Most of the data sets processed by VisIt come from parallelized simulation codes. In order to run in parallel, these codes decompose their data into pieces, called *domains*. The domain decomposition is chosen so that the total surface area of the boundaries between domains is minimized, and there is typically one domain for each processor. Also, when these codes write their data to disk, it is typically written in its domain decomposed form. Reading in one domain from these files is usually an atomic operation; the data is laid out such that either it is not possible or it is not advantageous to read in a portion of the domain.

Some data sets provide meta-data to VisIt, allowing VisIt to speed up their processing. We define meta-data to be data about the data set that is much smaller in size than the whole data set. Examples of meta-data are spatial extents for each domain of the simulation or variable extents for each domain for a specific variable of the simulation.

VisIt leverages the domain decomposition of the simulation for its own parallelization model. The number of processors that VisIt's parallel server is run on is typically much less than the number of domains the simulation code produced. So VisIt must support *domain overloading*, where multiple domains are processed on each processor of VisIt's server. Note that it is not sufficient to simply combine unrelated domains into one larger domain. This is not even possible for some grid types, like rectilinear grids where two grids are likely not neighboring and cannot be combined. And for situations where grids can be combined, like with unstructured grids, additional overhead would be incurred to distinguish which domains the elements in the combined grid originated from, which is important for operations where users want to refer to their elements in their original form (for example, picking elements).

The simulation data VisIt handles is some of the biggest ever produced. For example, VisIt was recently used to interactively visualize a data set comprised of 12.7 billion elements per time step using only eighty processors. In addition, there are typically on the order of one thousand time steps for each simulation. The grids are unstructured, structured, or rectilinear. There are also scattered data and structured Adaptive Mesh Refinement (AMR) grids.

2.2 Related Work

VisIt's base data flow network system is similar to those implemented in many other systems, for example VTK [11], OpenDX [1], and AVS [13]. The distinguishing feature of VisIt's data flow networks is the contract that enables optimizations to be applied adaptively. It should be noted that VisIt makes heavy use of VTK [6] modules to perform certain operations. Many of VisIt's data flow network components satisfy their *execute* phase by offloading work to VTK modules. But VisIt's abstract data flow network components remain distinct from VTK and, moreover, have no knowledge of VTK.

There are several other richly-featured parallel visualization tools that perform data reduction in parallel followed by a combined rendering stage, although these tools frequently do not support operating on the data in its original form (including domain overloading) in conjunction with collective communication. Examples of these are EnSight [4], ParaView [7], PV3 [5] and FieldView [8].

The concept of reading in only chunks of a larger data set (See Section 3.1) has been well discussed, for example by Chiang, et al. [3] and Pascucci et al. [10]. But these approaches typically do not support operating on the data in its native, domain decomposed form nor operating at the granularity of its atomic read operations (i.e. domains).

One of VisIt's execution models, called *streaming* (See Section 3.2), maps well to out-of-core processing. Many out-of-core algorithms are summarized by Silva et al. [12]. In addition, Ahrens et al. [2] gives an overview of a parallel streaming architecture. It should be noted that VisIt's streaming is restricted to domain granularity, while the system described by Ahrens allows for finer granularity. In this paper, the discussion will be limited to deciding when streaming is a viable technique and how the contract-based system enables VisIt to automatically choose the best execution model for a given pipeline.

Ghost elements are typically created by the simulation code and stored with the rest of the data. The advantages of utilizing ghost elements to avoid artifacts at domain boundaries (See Section 3.3) were discussed in [2]. In this paper, we propose that the post-processing tool (e.g. VisIt) be used to generate ghost data when ghost data is not available in the input data. Further, we discuss the factors that require when and what type of ghost data should be generated, as well as a system that can incorporate these factors (i.e. contracts).

3 Optimizations

In the following sections, some of the optimizations employed by VisIt will be described. The potential application of these optimizations is dependent on the properties of a pipeline's components. VisIt's contract-based system is necessary to facilitate these optimizations being applied adaptively.

After the optimizations are described, a complete description of VisIt's contract-based system will be presented.

3.1 Reading the Optimal Subset of Data

I/O is the most expensive portion of a pipeline execution for almost every operation VisIt performs. VisIt is able to reduce the amount of time spent in I/O to a minimum by reading only the domains that are relevant to any given pipeline. This performance gain propagates through the pipeline, since the domains not read in do not have to be processed downstream.

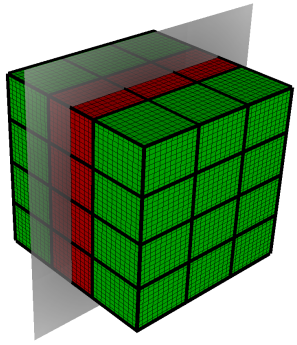

Figure 2: Shown is a 36 domain data set. The domains have thick black lines and are colored red or green. Mesh lines for the elements are also shown. To create the data set sliced by the transparent grey plane, only the red domains need to be processed. The green domains can be eliminated before ever being read in.

Consider the example of slicing a three-dimensional data set by a plane (see Figure 2). Many of the domains will not intersect the plane and reading them in will be wasted effort. In fact, the number of domains (D) that are intersected by the slice is typically $O(D^{2/3})$.

With the presence of meta-data, it is possible to eliminate domains from processing before ever reading them. For example, if the slice filter had access to the spatial extents for each domain, it could calculate the list of domains whose bounding boxes intersects the slice and only process that list (note that false positives can potentially be generated by considering only the bounding box).

VisIt's contract methodology enables this. During the update phase, every filter is given an opportunity to modify the contract, which contains the list of domains to be processed. A filter can check to see if some piece of meta-data is available (for example, spatial extents), and, if so, cross-reference the list of domains to be processed with the meta-data. The modified contract will then contain only those domains indicated by the filter.

It is important to note that every filter in the pipeline has a chance to modify the contract. If a pipeline had a slice filter and a contour filter (to generate isolines), the slice filter could use a spatial extents meta-data object to get exactly the set of domains that intersected the slice, while the contour filter could use a data extents meta-data object to get exactly the set of domains that could possibly produce contours. The resulting contract would contain the intersection of their two domain lists.

Further, since the infrastructure for subsetting the data is encapsulated in the contract, plugin filters can leverage this optimization. For example, a plugin spherical-slice filter can be added afterwards and it can also make use of the spatial extents meta-data, or a plugin filter that thresholds the data to produce only the elements that meet a certain criteria (elements with density between 2 g/cc and 5 g/cc, for example) can use the data extents meta-data. Also, the types of meta-data incorporated are not limited to spatial and data extents. They can take any form and can be arbitrarily added by new plugin developers.

3.2 Execution Model

VisIt has two techniques to do domain overloading. One approach, called *streaming*, will process domains one at a time. In this approach, there is one pipeline execution for each domain. Another approach, called *grouping*, is to execute the pipeline only once and to have each component process all of the domains before proceeding to the next one.

VisIt can employ either streaming or grouping when doing its load balancing. With *static load balancing*, domains are assigned to the processors at the beginning of the pipeline execution and a grouping strategy is applied. Because all of the data is available at every stage of the pipeline, collective communication can take place, enabling algorithms that cannot be efficiently implemented in an out-of-core setting. With *dynamic load balancing*, domains are assigned dynamically and a streaming strategy is applied. Not all domains take the same amount of time to process; dynamic load balancing efficiently (and dynamically) schedules these domains, creating an evenly distributed load. In addition, this strategy will process one domain in entirety before moving on to the next one, increasing cache coherency. However, because the data streams through the pipeline, it is not all available at one time and collective communication cannot take place with dynamic load balancing.

So how does VisIt decide which load balancing method to use? Dynamic load balancing is more efficient when the amount of work per domain varies greatly, but the technique does not support all algorithms. Static load balancing is usually less efficient, but does support all algorithms. The best solution is to use dynamic load balancing when possible, but fall back on static load balancing when an algorithm can not be implemented in a streaming setting. VisIt's contract system is again used to solve this problem. When each pipeline component gets the opportunity to modify the contract, it can specify whether or not it will use collective communication. When the load balancer executes, it will consult the contract and then use that information to adaptively choose between dynamic and static load balancing.

3.3 Generation of Ghost Data

Although handling the data set as individual domains is a good strategy, problems can arise along the exterior layer of elements of a domain that would not occur if the data set was processed as a single, monolithic domain.

One common operation is to remove a portion of a data set (for example, clipping a wedge out of a sphere) and then look at only the external faces of what remains. This can be done by finding the external faces of each of the data set's domains. But faces that are external to a domain can be internal to the whole data set. These extra faces can have multiple negative impacts. One impact is that the number of triangles being drawn can go up by an order of magnitude. Another impact occurs when the external faces are rendered transparently. Then the extra faces are visible and result in an incorrect image (See Figure 3).

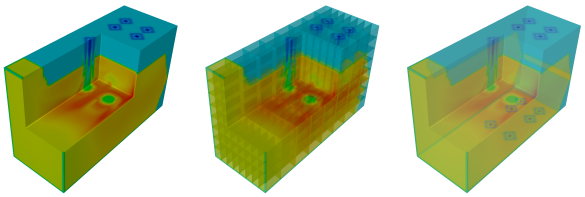

Figure 3: On the left is an opaque picture of the data set. In the middle, the opacity has been lowered. Faces external to a domain (yet internal to the data set) are being rendered. On the right, the faces have been removed. There are 902,134 triangles for the middle surface and only 277,796 for the right surface.

Now consider the case where interpolation is needed to perform a visualization operation. For example, consider the case where a contour is to be calculated on a data set that has an element-centered scalar quantity defined on it. Since contouring is typically done with an algorithm that requires node-centered data, the first step of this process is to interpolate the data to be a node-centered quantity from an element-centered quantity. Along the domain boundaries, the interpolation will be incorrect, because the elements from neighboring domains are not available. Ultimately, this will lead to a cracked contour surface (See Figure 4).

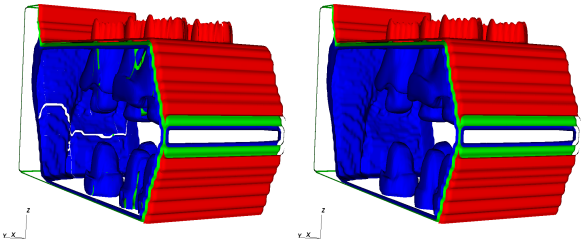

Figure 4: On the left is a contour plot of an element-centered quantity where ghost elements were not generated. Cracks in the contour surface occur along domain boundaries. On the right, ghost elements were generated and the correct picture was generated.

Both of the above problems require ghost data. For the first case, it is sufficient to mark the exterior faces of a domain that are internal to the whole data set as ghost faces. Then these ghost faces can be

discarded when an external face list is generated. The second case requires a redundant layer of ghost elements around the exterior of each domain. This allows interpolation to be done correctly.

Generation of ghost data is typically possible given some description of the input data set. This input can take several forms. One form can be a description of how the domain boundaries of structured meshes overlap ("faces I=0-5, J=0, K=8-12 of domain 5 are the same as faces I=12-17, J=17, K=10-14 of domain 12"). Another form utilizes global node identifiers assigned by the simulation code for each node in the problem. The visualization tool can then use these identifiers to determine which nodes are duplicated on multiple domains and thus identify shared boundaries between domains, which is the key step for creating ghost data. A third form uses the spatial coordinates of each node as a surrogate for global node identifiers. For each of these forms, the salient issue is that a module can be written where VisIt can give the module the input data and request it to create ghost faces or ghost elements. The details of such a module are not important to this paper.

There are costs associated with ghost data. Routines to generate ghost elements are typically implemented with collective communication, which precludes dynamic load balancing. In addition, ghost elements require a separate copy of the data set (see Figure 5), increasing memory costs. Ghost faces are less costly, but still require arrays of problem size data to track which faces are ghost and which are not. To this end, it is important to determine the exact type of ghost data needed, if any.

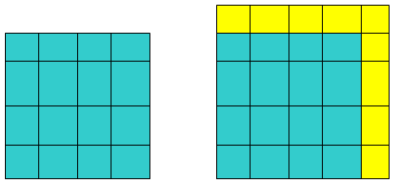

Figure 5: On the left is a domain without ghost elements. On the right is the same domain with ghost elements added (drawn in yellow). VisIt combines the ghost elements with the domain's real elements into one large domain for efficiency purposes for the filters downstream as well as simplicity of coding.

VisIt's contract system ensures that the minimum calculation is performed. If a filter, such as the contour filter, needs to do interpolation, it will mark the contract with this information on the update phase. As a result, ghost elements will be created at the top of the pipeline, allowing correct interpolation to occur. If the filter believes it will have to calculate external face lists, the case with the external face list filter, then it will mark the contract with this information on the update phase, and ghost faces will be created. Most importantly, in cases that do not require ghost data, such as the case when a data set is being sliced or volume rendered, no ghost data will be created.

3.4 Subgrid Generation

Before discussing subgrid generation, first consider VisIt's Clip and Threshold filters. Clipping allows a user to remove portions of a data set based on standard geometric primitives, such as planes or spheres. Thresholding allows the user to generate a data set where every element meets a certain criteria – the elements where density is greater than 2 g/cc and the temperature is between six hundred and eight hundred degrees Celsius. Both of these filters produce unstructured grid outputs even if the input grid is structured.

Our experience has been that most simulations with the largest number of elements are performed on rectilinear grids. Rectilinear grids have an implicit representation that minimizes the memory footprint of a data set. Many filters in VisIt, such as Clip and Threshold, take in rectilinear grid inputs and create unstructured grid outputs. One issue with the unstructured grid outputs is that many components have optimized routines for dealing with rectilinear grids. A bigger issue is that of memory footprint. The representation of an unstructured grid is explicit, and the additional memory required to store them can be more than what is available on the machine.

To further motivate this problem, consider the following example of a ten billion element rectilinear grid with a scalar, floating-point precision variable. The variable will occupy forty gigabytes (40GB) of memory. The representation of the grid itself takes only a few thousand bytes. But representing the same data set as an unstructured grid is much more costly. Again, the scalar variable will take forty gigabytes. Each element of the grid will now take eight integers to store the indices of the element's points in a point list, and each point in the point list will now take three floating-point precision numbers, leading the total memory footprint to be approximately four hundred eighty gigabytes (480GB). Of course, some operations dramatically reduce the total element count – a threshold filter applied to a ten billion element rectilinear grid may result in an unstructured grid consisting of just a few elements. In this case, the storage cost for an unstructured grid representation of the filter's output is insignificant when compared to the cost of the filter's input. However, the opposite can also happen: a threshold filter might remove only a few elements, creating an unstructured grid that is too large to store in memory.

VisIt addresses this problem by identifying complete rectilinear grids in the filter's output. These grids are then separated from the remainder of the output and remain as rectilinear grids. Accompanying this grid is one unstructured grid that contains all of the elements that could not be placed in the output rectilinear grids. Of course, proper ghost data is put in place to prevent artificial boundary artifacts from appearing (which type of ghost data is created is determined in the same way as described in Section 3.3). This process is referred to as *subgrid generation* (See Figure 6).

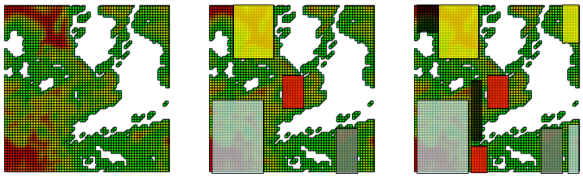

Figure 6: On the left, there is rectilinear grid with portions removed. In the middle, we see a covering with a large minimum grid size, which results in four grids. On the right, we see a covering with a smaller minimum grid size, which results in nine grids. The elements not covered in the output grids are placed in an unstructured grid.

There are many different configurations where rectilinear grids can be overlaid onto the "surviving elements" in the unstructured grid. The best configuration would maximize the number of elements covered by the rectilinear grids and minimize the total number of rectilinear grids. These are often opposing goals. Each element could be covered by simply devoting its own rectilinear grid to it. Since each grid has overhead, that would actually have a higher memory footprint than storing them in the original unstructured grid, defeating the purpose.

Although our two goals are opposing, we are typically more interested in one goal than another. For example, if we are trying to volume render the data set, then the performance of the algorithm is far superior on rectilinear grids than on unstructured grids. In this case, we would want to make sure the maximum number of elements was covered by the rectilinear grids. But the performance of many operations is indifferent to grid type, making memory foot-

print the only advantage for those operations.

VisIt uses its contract-based system to guide the placement of the rectilinear grids. Each component can modify the contract to say which goal it values - solely minimizing memory footprint versus keeping as many elements as possible in a native rectilinear representation for further processing.

As previously mentioned, some complete rectilinear grids contain so few elements that leaving them in an implicit form does not provide a favorable memory tradeoff, because there is overhead associated with each rectilinear grid. As such, VisIt has a minimum grid size of 2048 elements when overlaying complete rectilinear grids on the output. However, if the contract reports that filters down stream can take advantage of rectilinear representations, then the minimum grid size drops to 256 elements.

4 Description of Contract

The contract is simply a data structure. An initial version is created at the sink with all default values. As each component of the pipeline is visited during the update phase, it can modify the contract by changing members of this data structure. Table 1 contains a description of the members of VisIt's contract referenced in the previous sections.

Name	Type	Default Value
ghostType	enum {None Face, Element}	None
optimizedFor-Rectilinear	bool	false
canDoDynamic	bool	true
domains	vector<bool>	all true

Table 1: The members of VisIt's contract data structure described in previous sections.

Each filter in VisIt inherits from a base class, called *avtFilter*. This class has a virtual function that allows the filter to modify the contract. Below is pseudocode for how to modify a contract.

```
void avtXYZFilter::ModifyContact(avtContract *c)
{
   c->SetCanDoDynamic(false);
   c->SetGhostType(Element);
}
```

The contract allows for each component to describe the impact it will have on the total pipeline execution in a general way that does not require knowledge of the component itself. By enumerating the impacts that components can have on a pipeline, VisIt delivers a system that can be easily extended with new components. In addition, by using inheritance, the burden to implement a new component and utilize the contract-based system is very low.

The *ghostType* data member is set to *None* by default, because ghost data should not be generated when it is not required. The contour filter modifies the contract to have *Element* when it is going to contour element-based data, whereas the external face list filter modifies the contract to have *Face*. It should be noted that all of the components that modify this field do not blindly assign their desired value to it. If the face list filter were to overwrite a value of *Element* with its desired *Face*, then the filters downstream would not get the ghost data it needs. In this case, there is an ordering between the types. If *Element* was requested, then it should be obtained, regardless of requests for *Face* data. Similarly, *Face* data should be obtained even if other filters need *None*. And those that request *None* should gracefully accept and pass through ghost data. Furthermore, those that request *Face* data should be able to accept and deal gracefully with *Element* data in its place. The external face list filter, then, is able to accommodate *Element* data, even when it simply needs *Face* data. Although it would be possible to eliminate this complexity (by having separate entries in the contract for ghost faces and ghost elements), the field is maintained as one entry because it is more efficient to only calculate one set of ghost data.

optimizedForRectilinear is set to false by default, since only certain filters are specialized for operating on rectilinear grids. If the field is false, then the grids are placed to minimize memory footprint, rather than maximizing the number of elements covered by the rectilinear grids. *canDoDynamic* is set to true because it assumed that most filters do not require collective communication. If they do require collective communication, it is their responsibility to set that *canDoDynamic* to false when it has a chance to modify the contract in the update phase. Finally all of the *domains* are assumed to be used at the beginning of the update. If filters are able to access meta-data and determine that some domains will not affect the final picture, then they may modify the Boolean values for those domains.

5 Results

The contract-based system described in this paper has been fully implemented in VisIt. This includes of all of VisIt's components, which modify the contract as appropriate on update requests.

The results presented in this section demonstrate the benefit of the example optimizations discussed. We believe that this motivates the importance of using these optimizations and, by extension, motivates the importance of a contract-based system that enables these specialized optimizations to be adaptively employed.

We will present results in the context of a Rayleigh-Taylor Instability simulation, which models fluid instability between heavy fluid and light fluid. The simulation was performed on a 1152x1152x1152 rectilinear grid, for a total of more than one and a half billion elements. The data was decomposed into 729 domains, with each domain containing more than two million elements.

All timings were taken on Lawrence Livermore National Laboratory's *Thunder* machine, which was ranked seventh on the Top 500 list released in June 2005. The machine is comprised of 4096 1.4GHz Intel Itanium2 processors, each with access to two gigabytes of memory. The machine is divided into 1024 nodes, where each node contains four processors. The processor's memory can only be shared with other processors in its node.

Pictures of the operations described below are located at the end of this paper.

5.1 Reading the optimal subset of data

We will present two algorithms where the optimal subset of data was read – slicing, which makes use of spatial meta-data, and contouring, which makes use of variable meta-data. It should be noted that use of spatial meta-data typically yields a consistent performance improvement, but performance improvement from variable meta-data can be highly problem specific. To illustrate this, results from early in the simulation and late in the simulation are shown (See Table 2). The processing time includes the time to read in a data set from disk, perform operations to it, and prepare it for rendering. Rendering was not included because it can be highly dependent on screen size.

5.2 Comparison of execution models

Since not all pipelines can successfully execute with dynamic load balancing, we can only compare execution time for those pipelines

Algorithm	Processors	Processing time (sec) Without Meta-data	With Meta-data
Slicing	32	25.3	3.2
Contouring (early)	32	41.1	5.8
Contouring (late)	32	185.0	97.2

Table 2: Measuring effectiveness of reading the optimal subset of data

that can use dynamic load balancing. Again using the Rayleigh-Taylor Instability simulation, we study the performance of slicing, contouring, thresholding, and clipping. Note that other optimizations were used in this study – slicing and contouring were using spatial and variable meta-data respectively, while thresholding and clipping used subgrid generation for its outputs (See Table 3). Again, the processing time includes the time to read in a data set from disk, perform operations to it, and prepare it for rendering.

Algorithm	Processors	Processing time (sec) Static LB	Dynamic LB
Slicing	32	3.2	4.0
Contouring	32	97.2	65.1
Thresholding	64	181.3	64.1
Clipping	64	59.0	30.7

Table 3: Measuring performance differences between static and dynamic load balancing

Slicing did not receive large performance improvements from dynamic load balancing, because our use of spatial meta-data eliminated those domains not intersecting the slice, and the amount of work performed per processor was relatively even. We believe that the higher dynamic load balancing time is due to the overhead in multiple pipeline executions. Contouring, thresholding, and clipping, on the other hand, did receive substantial speedups, since the time to execute each of these algorithms was highly dependent on its input domains.

5.3 Generation of ghost data

This optimization is not a performance optimization; it is necessary to create the correct picture. Hence, no performance comparisons are presented here. Refer back to Figures 3 and 4 in Section 3.3 to see the results.

5.4 Subgrid Generation

VisIt's volume renderer processes data in three phases. The first phase samples the data along rays. The input data can be heterogeneous, made up of both rectilinear and unstructured grids. The rectilinear grids will be sampled quickly using specialized algorithms, while the unstructured grids will be sampled slowly using generalized algorithms. The sampling done on each processor uses the data assigned to that processor by the load balancer. Once the sampling has been completed, the second phase, a communication phase, begins. During this phase, samples are re-distributed among processors, to prepare for the third phase, a compositing phase. The compositing is done on a per-pixel basis. Each processor is responsible for compositing some portion of the screen, and the second, communication phase, brings the samples necessary to perform this operation.

The volume renderer uses the contract to indicate that it has rectilinear optimizations. This will cause the subgrid generation module to create more rectilinear grids, many of them smaller in size than what is typically generated. This then allows the sampling phase to use the specialized, efficient algorithms and finish much more quickly.

In the results below, we list the time to create one volume rendered image. Before volume rendering, we have clipped the data set or thresholded the data set and used subgrid generation to create the output. Table 4 measures the effectiveness of allowing for control of the minimum grid size (2048 versus 256) with subgrid generation. When subgrid generation was not used, only unstructured grids were created, and these algorithms exhausted available memory, leading to failure with this number of processors.

It should be noted that the rendering time is dominated by sampling the unstructured grids. This data set can be volume rendered in 0.25 seconds when no operations (such as clipping or thresholding) are applied to it.

Algorithm	Processors	Subgrid Generation No	Yes Minimum Grid Size 2048	256
Clip	64	Out Of Memory	12.0s	9.0s
Thresholding	64	Out Of Memory	11.4s	10.8s

Table 4: Measuring effectiveness of grid size control with subgrid generation

The thresholded volume rendering produces only marginal gains, since the fluids have become so mixed that even rectilinear grids as small as 256 elements cannot be placed over much of the mixing region.

6 Conclusion

The scale of the data being processed by VisIt requires that as many optimizations as possible be included in each pipeline execution. Yet the tool's large number of components, including the addition of new plugin components, makes it difficult to determine which optimizations can be applied. VisIt's contract-based system solves this problem, allowing all possible optimizations to be applied to each pipeline execution. The contract is a data structure that extends the standard data flow network design. It provides a prescribed interface that every pipeline component can modify. Furthermore, the system is extensible, allowing for further optimizations to be added and supported by the contract system.

This system has been fully implemented and deployed to users. VisIt is used hundreds of times daily and has a user base of hundreds of people.

7 Acknowledgements

VisIt has been developed by B-Division of Lawrence Livermore National Laboratory and the Advanced Simulation and Computing Program (ASC). This work was performed under the auspices of the U.S Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. Lawrence Livermore National Laboratory, P.O. Box 808, L-159, Livermore, Ca, 94551

REFERENCES

[1] Greg Abram and Lloyd A. Treinish. An extended data-flow architecture for data analysis and visualization. Research report RC 20001 (88338), IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, February 1995.

[2] James Ahrens, Kristi Brislawn, Ken Martin, Berk Geveci, C. Charles Law, and Michael Papka. Large-scale data visualization using parallel data streaming. *IEEE Comput. Graph. Appl.*, 21(4):34–41, 2001.

[3] Yi-Jen Chiang and Claudio T. Silva. I/o optimal isosurface extraction (extended abstract). In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 293–ff. IEEE Computer Society Press, 1997.

[4] Computational Engineering International, Inc. *EnSight User Manual*, May 2003.

[5] R. Haimes and D. Edwards. Visualization in a parallel processing environment, 1997.

[6] Kitware, Inc. *The Visualization Toolkit User's Guide*, January 2003.

[7] C. Charles Law, Amy Henderson, and James Ahrens. An application architecture for large data visualization: a case study. In *PVG '01: Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, pages 125–128. IEEE Press, 2001.

[8] Steve M. Legensky. Interactive investigation of fluid mechanics data sets. In *VIS '90: Proceedings of the 1st conference on Visualization '90*, pages 435–439. IEEE Computer Society Press, 1990.

[9] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *IEEE Comput. Graph. Appl.*, 14(4):23–32, 1994.

[10] Valerio Pascucci and Randall J. Frank. Global static indexing for real-time exploration of very large regular grids. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 2–2. ACM Press, 2001.

[11] William J. Schroeder, Kenneth M. Martin, and William E. Lorensen. The design and implementation of an object-oriented toolkit for 3d graphics and visualization. In *VIS '96: Proceedings of the 7th conference on Visualization '96*, pages 93–ff. IEEE Computer Society Press, 1996.

[12] C. Silva, Y. Chiang, J. El-Sana, and P. Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics. In *Visualization 2002 Course Notes*, 2002.

[13] Craig Upson, Thomas Faulhaber Jr., David Kamins, David H. Laidlaw, David Schlegel, Jeffrey Vroom, Robert Gurwitz, and Andries van Dam. The application visualization system: A computational environment for scientific visualization. *Computer Graphics and Applications*, 9(4):30–42, July 1989.

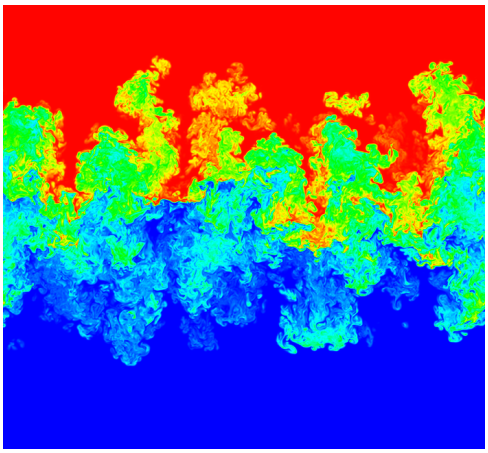

Figure 7: This is a slice of the simulation at late time. Light fluid is colored blue, heavy fluid is colored red.

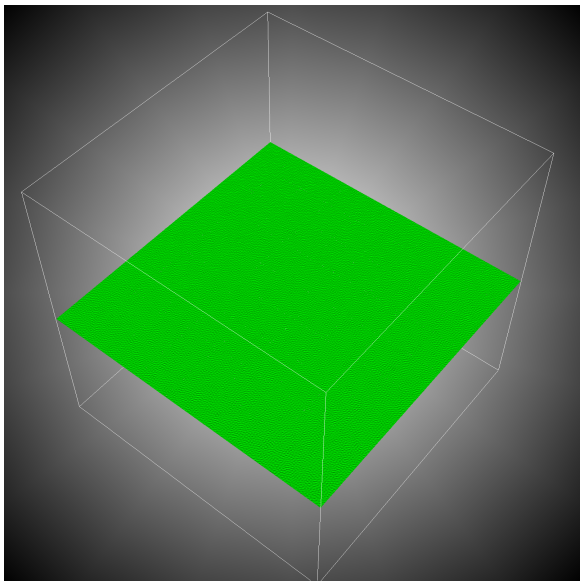

Figure 8: A contour at early simulation time. This contour separates the light and dense fluids.

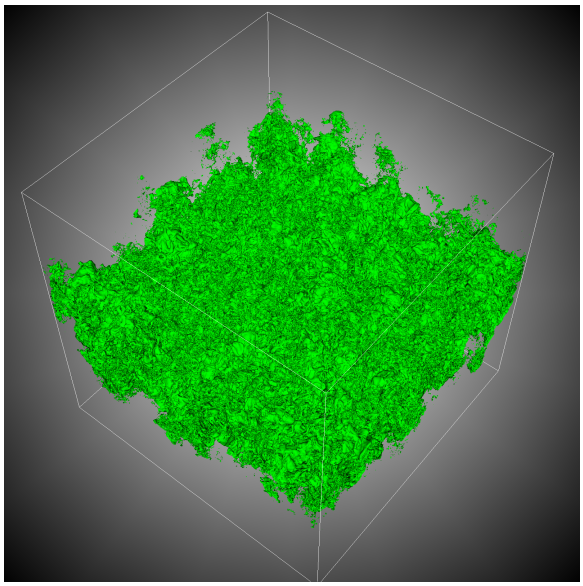

Figure 9: A contour at late simulation time. This contour separates the light and dense fluids.

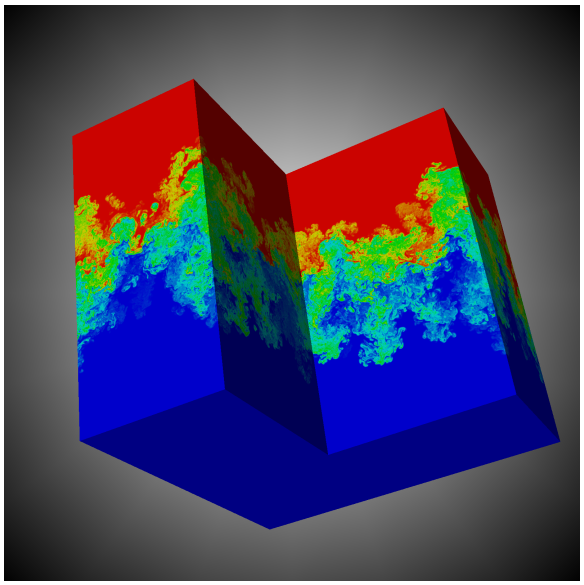

Figure 10: Pictured here is the simulation with one portion clipped away. Light fluid is colored blue, heavy fluid is colored red.

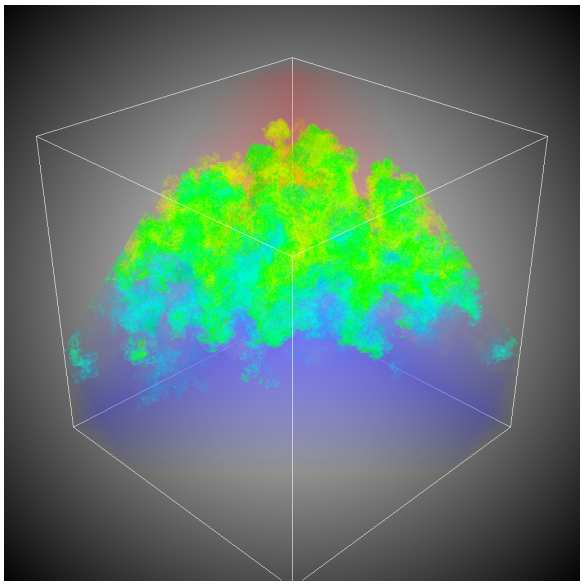

Figure 12: Pictured here is a volume rendering of the simulation after being clipped by a plane.

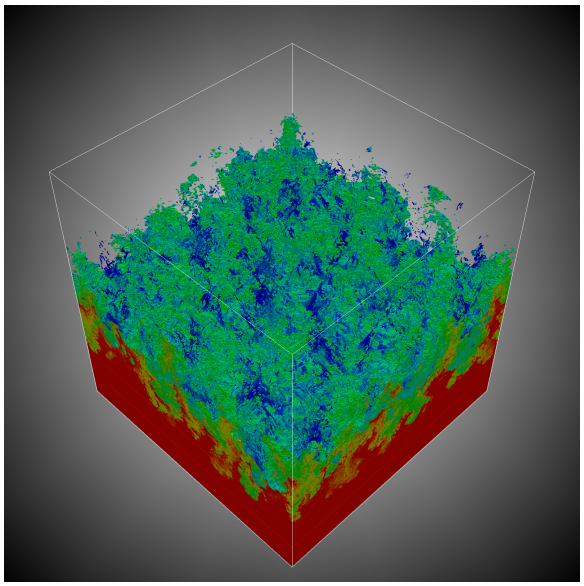

Figure 11: Pictured here is the simulation with the light fluid removed using the threshold operation.

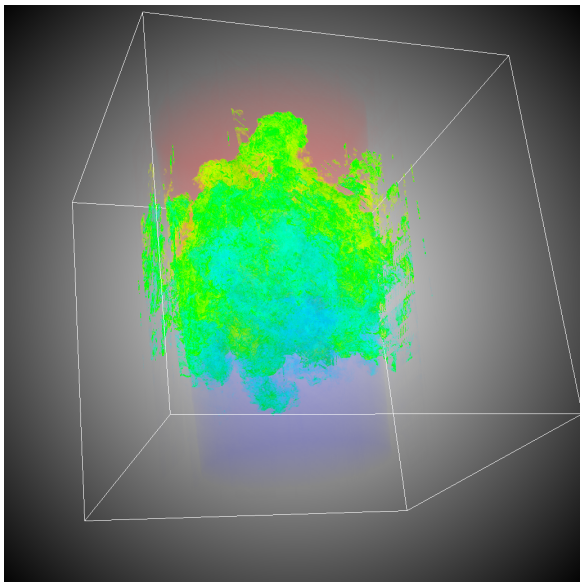

Figure 13: This is a volume rendering of the simulation after thresholding by a distance variable to remove elements outside a cylinder.